

Types of Software Design Patterns

There are three types of Design Patterns:

- Creational Design Pattern
- Structural Design Pattern
- Behavioral Design Pattern

1. Creational Design Patterns

Creational Design Patterns focus on the process of object creation or problems related to object creation. They help in making a system independent of how its objects are created, composed and represented.



Singleton Method Design Pattern

- Of all, the Singleton Design pattern is the most straightforward to understand.
- It guarantees that a class has just one instance and offers a way to access it globally.

```
class Singleton {
    private static volatile Singleton instance; // Ensuring visibility across threads

    private Singleton() {
        // Private constructor to prevent instantiation
    }

    public static Singleton getInstance() {
        if (instance == null) { // First check
            synchronized (Singleton.class) { // Locking for thread safety
                if (instance == null) { // Second check
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }

    public void showMessage() {
        System.out.println("Singleton instance is working!");
    }
}

public class SingletonExample {
    public static void main(String[] args) {
        Singleton singleton1 = Singleton.getInstance();
        Singleton singleton2 = Singleton.getInstance();

        singleton1.showMessage();

        // Checking if both instances are the same
        System.out.println(singleton1 == singleton2); // true
    }
}
```

Factory Design Patter:

A Factory Pattern or Factory Method Pattern says that just define an interface or abstract class for creating an object but let the subclasses decide which class to instantiate. In other words, subclasses are responsible to create the instance of the class.

Example: Vehicle Factory

Imagine a system where we need to create **different types of vehicles** (Car, Bike, etc.). Instead of manually instantiating objects, we use a factory method.

Step 1: Define the Interface

Java

Copy

```
interface Vehicle {  
    void drive();  
}
```

Step 2: Create Concrete Implementations

Java

Copy

```
class Car implements Vehicle {  
    @Override  
    public void drive() {  
        System.out.println("Driving a car ...");  
    }  
}  
  
class Bike implements Vehicle {  
    @Override  
    public void drive() {  
        System.out.println("Riding a bike ...");  
    }  
}
```

Step 3: Implement Factory Class

Java

Copy

```
class VehicleFactory {  
    public static Vehicle getVehicle(String type) {  
        if ("car".equalsIgnoreCase(type)) return new Car();  
        if ("bike".equalsIgnoreCase(type)) return new Bike();  
        throw new IllegalArgumentException("Unknown vehicle type");  
    }  
}
```

Step 4: Use the Factory to Create Objects

Java

 Copy

```
public class FactoryPatternExample {  
    public static void main(String[] args) {  
        Vehicle car = VehicleFactory.getVehicle("car");  
        car.drive();  
  
        Vehicle bike = VehicleFactory.getVehicle("bike");  
        bike.drive();  
    }  
}
```

Abstract Factory Pattern:

The **Abstract Factory Pattern** is a **creational design pattern** used when you need to create **families of related objects** without specifying their concrete classes.



Use Case

You're building a system that supports multiple currencies. Based on the country, you want to get the **currency name** and **symbol** using an abstract factory.

◆ 2. Concrete Products

java

 Copy

 Edit

```
public class USDCurrency implements Currency {  
    public String getSymbol() {  
        return "USD $";  
    }  
}  
  
public class INRCurrency implements Currency {  
    public String getSymbol() {  
        return "INR ₹";  
    }  
}
```

◆ 3. Abstract Factory

```
java Copy Edit

public interface CurrencyFactory {
    Currency createCurrency();
}
```

◆ 4. Concrete Factories

```
java Copy Edit

public class USDCurrencyFactory implements CurrencyFactory {
    public Currency createCurrency() {
        return new USDCurrency();
    }
}

public class INRCurrencyFactory implements CurrencyFactory {
    public Currency createCurrency() {
        return new INRCurrency();
    }
}
```

◆ 5. Client Code

```
java Copy Edit

public class CurrencyApp {
    public static void main(String[] args) {
        CurrencyFactory factory;

        String country = "India"; // change to "USA" for USD

        if (country.equalsIgnoreCase("India")) {
            factory = new INRCurrencyFactory();
        } else {
            factory = new USDCurrencyFactory();
        }

        Currency currency = factory.createCurrency();
        System.out.println("Currency Symbol: " + currency.getSymbol());
    }
}
```

✓ Output (for "India")

```
javascript Copy Edit

Currency Symbol: INR ₹
```

Prototype Design Pattern:

The **Prototype Pattern** is a creational design pattern used to **clone objects** instead of creating new instances. This improves performance when object creation is expensive. It uses a clone() method to create a copy of an object

◆ 1. Define the Prototype Interface

java

Copy

Edit

```
public interface Prototype {  
    Prototype clone();  
}
```

◆ 2. Create a Concrete Class

java

Copy

Edit

```
public class Document implements Prototype {  
    private String title;  
    private String content;  
  
    public Document(String title, String content) {  
        this.title = title;  
        this.content = content;  
    }  
  
    public void show() {  
        System.out.println("Title: " + title);  
        System.out.println("Content: " + content);  
    }  
  
    @Override  
    public Prototype clone() {  
        return new Document(title, content);  
    }  
}
```

◆ 3. Use the Prototype

java

Copy

Edit

```
public class Main {  
    public static void main(String[] args) {  
        Document original = new Document("Prototype Pattern", "This is the original document.");  
        System.out.println("Original Document:");  
        original.show();  
  
        Document copy = (Document) original.clone();  
        System.out.println("\nCloned Document:");  
        copy.show();  
    }  
}
```

✓ Output

vbnet

Copy

Edit

```
Original Document:  
Title: Prototype Pattern  
Content: This is the original document.  
  
Cloned Document:  
Title: Prototype Pattern  
Content: This is the original document.
```

Builder Design Pattern:

The Builder Pattern is a creational design pattern that lets you construct complex objects step by step. It separates the construction of an object from its representation.

```
class Car {
    private String engine;
    private int wheels;
    private boolean airConditioning;

    private Car(CarBuilder builder) {
        this.engine = builder.engine;
        this.wheels = builder.wheels;
        this.airConditioning = builder.airConditioning;
    }

    @Override
    public String toString() {
        return "Car with " + engine + " engine, " + wheels + " wheels, AC: " + airConditioning;
    }

    public static class CarBuilder {
        private String engine;
        private int wheels;
        private boolean airConditioning;

        public CarBuilder setEngine(String engine) {
            this.engine = engine;
            return this;
        }

        public CarBuilder setWheels(int wheels) {
            this.wheels = wheels;
            return this;
        }

        public CarBuilder setAirConditioning(boolean airConditioning) {
            this.airConditioning = airConditioning;
            return this;
        }

        public Car build() {
            return new Car(this);
        }
    }
}
```

Step 2: Using the Builder Pattern

Java

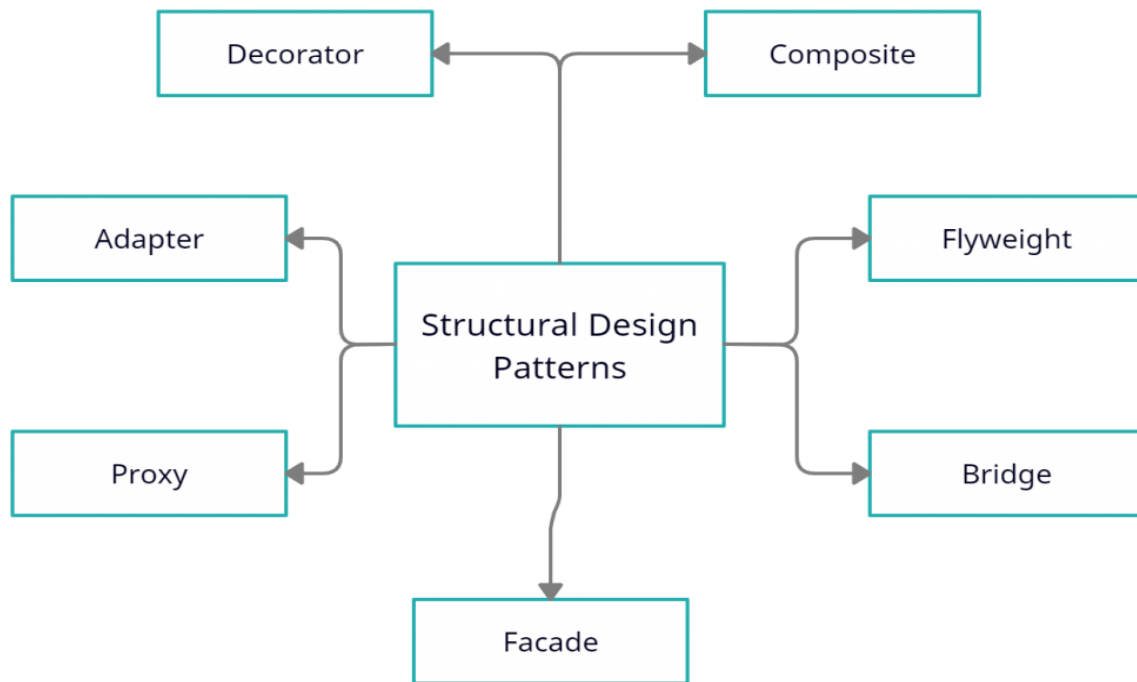
 Copy

```
public class BuilderPatternExample {
    public static void main(String[] args) {
        Car car = new Car.CarBuilder()
            .setEngine("V8")
            .setWheels(4)
            .setAirConditioning(true)
            .build();

        System.out.println(car);
    }
}
```

Structural Design Patterns:

The structural design patterns simplify the structure by identifying the relationships. Structural design patterns are concerned with how classes and objects can be composed



Adaptor Design Pattern:

The **Adapter Pattern** is a **structural design pattern** that allows two incompatible interfaces to work together **without modifying their source code**.

Example: Charging a Phone with Different Chargers

Imagine a scenario where you have a **new phone** that requires a **USB-C charger**, but you only have an **old Micro-USB charger**. Instead of replacing the charger, we create an **adapter** to make them compatible.

Step 1: Define the Target Interface (Modern Charger)

Java

Copy

```
interface USB_C_Charger {  
    void charge();  
}
```

Step 2: Define the Incompatible Class (Old Charger with Micro-USB Port)


Java

 Copy

```
class MicroUSBCharger {  
    void oldCharge() {  
        System.out.println("Charging with Micro-USB");  
    }  
}
```

Step 3: Create the Adapter to Bridge the Difference


Java

 Copy

```
class ChargerAdapter implements USB_C_Charger {  
    private MicroUSBCharger microUSBCharger;  
  
    public ChargerAdapter(MicroUSBCharger microUSBCharger) {  
        this.microUSBCharger = microUSBCharger;  
    }  
  
    @Override  
    public void charge() {  
        microUSBCharger.oldCharge(); // Converts Micro-USB charging to USB  
    }  
}
```


Step 4: Using the Adapter in Client Code

Java

 Copy

```
public class AdapterPatternExample {  
    public static void main(String[] args) {  
        MicroUSBCharger oldCharger = new MicroUSBCharger();  
        USB_C_Charger adapter = new ChargerAdapter(oldCharger); // Using a  
  
        adapter.charge(); // Works seamlessly  
    }  
}
```

✓ **Output:**

 Copy

```
Charging with Micro-USB
```

Bridge Design Pattern:

The **Bridge Pattern** is a **structural design pattern** that helps separate **abstraction** from its **implementation**, allowing changes independently.

Example: Device and Remote Control

Imagine a scenario where you have **different types of devices** (TV, Radio), and **different types of remote controls** (Basic, Advanced). Instead of tightly coupling them, we use **Bridge Pattern** to separate the abstraction (**Remote Control**) from the implementation (**Device**).

Step 1: Define the `Device` Interface

Java

 Copy

```
interface Device {  
    void powerOn();  
    void powerOff();  
}
```

Step 2: Implement Concrete Devices

```
class TV implements Device {  
    @Override  
    public void powerOn() {  
        System.out.println("TV is ON");  
    }  
  
    @Override  
    public void powerOff() {  
        System.out.println("TV is OFF");  
    }  
}  
  
class Radio implements Device {  
    @Override  
    public void powerOn() {  
        System.out.println("Radio is ON");  
    }  
  
    @Override  
    public void powerOff() {  
        System.out.println("Radio is OFF");  
    }  
}
```

Step 3: Define the `RemoteControl` Abstraction

Java

 Copy

```
abstract class RemoteControl {  
    protected Device device;  
  
    public RemoteControl(Device device) {  
        this.device = device;  
    }  
  
    abstract void turnOn();  
    abstract void turnOff();  
}
```

Step 4: Implement Concrete Remote Controls

```
Java Copy  
  
class BasicRemote extends RemoteControl {  
    public BasicRemote(Device device) {  
        super(device);  
    }  
  
    @Override  
    void turnOn() {  
        device.powerOn();  
    }  
  
    @Override  
    void turnOff() {  
        device.powerOff();  
    }  
}
```

Step 5: Using the Bridge Pattern in Client Code

```
Java Copy  
  
public class BridgePatternExample {  
    public static void main(String[] args) {  
        Device tv = new TV();  
        RemoteControl remote = new BasicRemote(tv);  
        remote.turnOn();  
        remote.turnOff();  
  
        System.out.println("-----");  
  
        Device radio = new Radio();  
        remote = new BasicRemote(radio);  
        remote.turnOn();  
        remote.turnOff();  
    }  
}
```

✓ Output:

```
Copy  
  
TV is ON  
TV is OFF  
-----  
Radio is ON  
Radio is OFF
```

Façade Design Pattern:

The **Facade Pattern** is a **structural design pattern** that provides a **simplified interface** to complex subsystems. It hides the internal complexity and makes a system easier to use.

Example: Home Theater System

Imagine a **home theater system** with multiple components (**TV, Speakers, DVD Player, etc.**). Instead of calling multiple methods for each device, we create a **Facade** that simplifies the user experience.

Step 1: Define Subsystem Components

```

class TV {
    public void turnOn() { System.out.println("TV is ON"); }
    public void turnOff() { System.out.println("TV is OFF"); }
}

class DVDPlayer {
    public void playMovie() { System.out.println("Playing DVD Movie"); }
    public void stopMovie() { System.out.println("Stopping DVD Movie"); }
}

class Speakers {
    public void setVolume(int level) { System.out.println("Setting Speaker Volume to " + level); }
}

```

Step 2: Create the Facade Class

```

class HomeTheaterFacade {
    private TV tv;
    private DVDPlayer dvdPlayer;
    private Speakers speakers;

    public HomeTheaterFacade() {
        this.tv = new TV();
        this.dvdPlayer = new DVDPlayer();
        this.speakers = new Speakers();
    }

    public void watchMovie() {
        System.out.println("Getting ready to watch a movie...");
        tv.turnOn();
        dvdPlayer.playMovie();
        speakers.setVolume(10);
    }

    public void endMovie() {
        System.out.println("Shutting down the home theater...");
        dvdPlayer.stopMovie();
        tv.turnOff();
    }
}

```

Step 3: Use Facade in Client Code

Java

 Copy

```

public class FacadePatternExample {
    public static void main(String[] args) {
        HomeTheaterFacade homeTheater = new HomeTheaterFacade();

        homeTheater.watchMovie();
        System.out.println("-----");
        homeTheater.endMovie();
    }
}

```

✓ **Output:**

 Copy

```

Getting ready to watch a movie ...
TV is ON
Playing DVD Movie
Setting Speaker Volume to 10
-----
Shutting down the home theater ...
Stopping DVD Movie
TV is OFF

```

Decorator Design Pattern:

The **Decorator Pattern** is a **structural design pattern** used to dynamically add behavior or responsibilities to objects **without modifying their structure**. It helps **extend object functionality** without altering the existing class.

Example: Coffee Customization

Imagine a coffee shop where you sell **different types of coffee**, and customers can **add extra toppings** like **milk, sugar, or caramel**. Instead of modifying the base

`Coffee` class, we use decorators to enhance the coffee dynamically.

Step 1: Define the Base Component (Coffee)

Java

 Copy

```
interface Coffee {  
    double getCost();  
    String getDescription();  
}
```

Step 2: Implement the Basic Coffee Class

Java

 Copy

```
class SimpleCoffee implements Coffee {  
    @Override  
    public double getCost() {  
        return 5.0; // Base price  
    }  
  
    @Override  
    public String getDescription() {  
        return "Simple Coffee";  
    }  
}
```

Step 3: Create Abstract Decorator Class

```
abstract class CoffeeDecorator implements Coffee {  
    protected Coffee coffee;  
  
    public CoffeeDecorator(Coffee coffee) {  
        this.coffee = coffee;  
    }  
  
    @Override  
    public double getCost() {  
        return coffee.getCost();  
    }  
  
    @Override  
    public String getDescription() {  
        return coffee.getDescription();  
    }  
}
```

Step 4: Implement Specific Decorators

```
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 2.0; // Extra cost for milk
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Milk";
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 1.0; // Extra cost for sugar
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Sugar";
    }
}
```

Step 5: Use Decorators in Client Code


```
public class DecoratorPatternExample {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());

        coffee = new MilkDecorator(coffee);
        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());

        coffee = new SugarDecorator(coffee);
        System.out.println(coffee.getDescription() + " - $" + coffee.getCost());
    }
}
```

✓ Output:

```
Simple Coffee - $5.0
Simple Coffee, Milk - $7.0
Simple Coffee, Milk, Sugar - $8.0
```

 Copy

Composite Design Pattern:

The **Composite Pattern** is a **structural design pattern** that treats **individual objects** and **groups of objects** uniformly. It helps represent **tree-like hierarchies**, such as a **file system**, **organizational structure**, or **UI components**

Example: Organization Hierarchy

Imagine a company where employees are structured in **departments**, and some employees **manage others**. The Composite Pattern allows treating **both individual employees and teams uniformly**.

Step 1: Define the Common Interface

Java

Copy

```
import java.util.ArrayList;
import java.util.List;

interface Employee {
    void showDetails();
}
```

Step 2: Implement Leaf Node (Individual Employee)

Java

Copy

```
class Developer implements Employee {
    private String name;
    private String role;

    public Developer(String name, String role) {
        this.name = name;
        this.role = role;
    }

    @Override
    public void showDetails() {
        System.out.println("Developer: " + name + ", Role: " + role);
    }
}
```

Step 3: Implement Composite Node (Manager)

```
class Manager implements Employee {
    private String name;
    private List<Employee> subordinates = new ArrayList<>();


    public Manager(String name) {
        this.name = name;
    }

    public void addEmployee(Employee emp) {
        subordinates.add(emp);
    }

    @Override
    public void showDetails() {
        System.out.println("Manager: " + name);
        for (Employee emp : subordinates) {
            emp.showDetails();
        }
    }
}
```

Step 4: Using the Composite Pattern in Client Code

Java

 Copy


```
public class CompositePatternExample {
    public static void main(String[] args) {
        Developer dev1 = new Developer("Alice", "Backend Developer");
        Developer dev2 = new Developer("Bob", "Frontend Developer");

        Manager teamLead = new Manager("Charlie");
        teamLead.addEmployee(dev1);
        teamLead.addEmployee(dev2);

        Manager director = new Manager("David");
        director.addEmployee(teamLead);

        System.out.println("Company Structure:");
        director.showDetails();
    }
}
```

✓ **Output:**

 Copy


```
Company Structure:
Manager: David
Manager: Charlie
Developer: Alice, Role: Backend Developer
Developer: Bob, Role: Frontend Developer
```

Proxy Design Pattern:

The **Proxy Pattern** is a **structural design pattern** that provides a surrogate or placeholder object to control access to another object. It is useful when you need **lazy initialization**, **security control**, or **logging before accessing a resource-heavy object**.

Step 1: Define the Common Interface


Java

 Copy

```
interface Database {
    void connect();
}
```

Step 2: Implement the Real Database Class

Java

 Copy


```
class RealDatabase implements Database {  
    @Override  
    public void connect() {  
        System.out.println("Connecting to the real database ...");  
    }  
}
```

Step 3: Implement the Proxy Class

```
class DatabaseProxy implements Database {  
    private RealDatabase realDatabase;  
    private String userRole;  
  
    public DatabaseProxy(String userRole) {  
        this.userRole = userRole;  
    }  
  
    @Override  
    public void connect() {  
        if ("Admin".equalsIgnoreCase(userRole)) {  
            if (realDatabase == null) {  
                realDatabase = new RealDatabase();  
            }  
            realDatabase.connect();  
        } else {  
            System.out.println("Access Denied: You do not have permission to connect to the database.")  
        }  
    }  
}
```


Step 4: Use Proxy in Client Code

Java

 Copy

```
public class ProxyPatternExample {  
    public static void main(String[] args) {  
        Database adminDatabase = new DatabaseProxy("Admin");  
        adminDatabase.connect();  
  
        System.out.println("_____");  
  
        Database guestDatabase = new DatabaseProxy("Guest");  
        guestDatabase.connect();  
    }  
}
```

✓ Output:

 Copy

Connecting to the real database ...

Access Denied: You do not have permission to connect to the database.

Fly Weight Design Pattern:


The **Flyweight Pattern** is a **structural design pattern** that minimizes **memory usage** by sharing **common objects** instead of creating **new instances**. It is particularly useful for applications with **large numbers of similar objects**, such as **text editors, gaming sprites, or caching mechanisms**.

Example: Creating a Character Factory for a Text Editor

A text editor **handles thousands of characters**. Instead of creating a **new object for each character**, Flyweight **reuses existing characters**.

Step 1: Create the Flyweight Interface


Java

 Copy

```
interface Character {  
    void display(String font);  
}
```

Step 2: Implement Concrete Flyweight (Reusable Objects)

Java

 Copy

```
class ConcreteCharacter implements Character {  
    private final char symbol;  
  
    public ConcreteCharacter(char symbol) {  
        this.symbol = symbol;  
    }  
  
    @Override  
    public void display(String font) {  
        System.out.println("Displaying '" + symbol + "' in font: " + font);  
    }  
}
```

Step 3: Implement Flyweight Factory

The factory **maintains a cache of characters** and returns existing instances **instead of creating new ones**.

```
import java.util.HashMap;  
import java.util.Map;  
  
class CharacterFactory {  
    private static final Map<Character, Character> characterCache = new HashMap<>();  
  
    public static Character getCharacter(char symbol) {  
        characterCache.putIfAbsent(symbol, new ConcreteCharacter(symbol));  
        return characterCache.get(symbol);  
    }  
}
```

Step 4: Using the Flyweight Pattern in Client Code

```
public class FlyweightPatternExample {  
    public static void main(String[] args) {  
        Character c1 = CharacterFactory.getCharacter('A');  
        c1.display("Arial");  
  
        Character c2 = CharacterFactory.getCharacter('B');  
        c2.display("Times New Roman");  
  
        Character c3 = CharacterFactory.getCharacter('A'); // Reuses 'A' object  
        c3.display("Courier");  
  
        System.out.println("Same object for 'A' characters? " + (c1 == c3)); // true  
    }  
}
```

✓ Output:

```
Displaying 'A' in font: Arial  
Displaying 'B' in font: Times New Roman  
Displaying 'A' in font: Courier  
Same object for 'A' characters? true
```

Copy

Behavioural Design Patterns:

Behavioural design patterns are a category of [design patterns](#) that focus on the interactions and communication between objects. They help define how objects collaborate and distribute responsibility among them, making it easier to manage complex control flow and communication in a system.



Behavioral Design Pattern



Chain of Responsibility Pattern:

The **Chain of Responsibility Pattern** is a **behavioral design pattern** that allows multiple handlers to process a request in **sequence** until one of them **handles it**. It's commonly used in **logging frameworks**, **authentication mechanisms**, **request validation**, and **event handling**.

Example: Logging System

Imagine a **logging system** where different levels of log messages (**INFO**, **DEBUG**, **ERROR**) need to be handled by appropriate loggers.

Step 1: Define the Logger Interface

Java

Copy

```
public void setNextLogger(Logger nextLogger) {
    this.nextLogger = nextLogger;
}

public void logMessage(String message, String level) {
    if (handleLog(message, level)) {
        return; // Stop processing if handled
    }
    if (nextLogger != null) {
        nextLogger.logMessage(message, level); // Pass to next handler
    }
}

protected abstract boolean handleLog(String message, String level);
}
```

Step 2: Implement Concrete Loggers

```
class InfoLogger extends Logger {
    @Override
    protected boolean handleLog(String message, String level) {
        if ("INFO".equalsIgnoreCase(level)) {
            System.out.println("[INFO]: " + message);
            return true;
        }
        return false;
    }
}

class DebugLogger extends Logger {
    @Override
    protected boolean handleLog(String message, String level) {
        if ("DEBUG".equalsIgnoreCase(level)) {
            System.out.println("[DEBUG]: " + message);
            return true;
        }
        return false;
    }
}

class ErrorLogger extends Logger {
    @Override
    protected boolean handleLog(String message, String level) {
        if ("ERROR".equalsIgnoreCase(level)) {
            System.out.println("[ERROR]: " + message);
            return true;
        }
        return false;
    }
}
```

Step 3: Build and Use the Chain

Java

 Copy

```
public class ChainOfResponsibilityExample {
    public static void main(String[] args) {
        // Creating loggers and linking them in chain
        Logger infoLogger = new InfoLogger();
        Logger debugLogger = new DebugLogger();
        Logger errorLogger = new ErrorLogger();

        infoLogger.setNextLogger(debugLogger);
        debugLogger.setNextLogger(errorLogger);

        // Processing messages
        infoLogger.logMessage("This is an informational message.", "INFO");
        infoLogger.logMessage("Debugging application flow.", "DEBUG");
        infoLogger.logMessage("An error occurred!", "ERROR");
    }
}
```

✓ **Output:**

 Copy

```
[INFO]: This is an informational message.
[DEBUG]: Debugging application flow.
[ERROR]: An error occurred!
```

Command Design Pattern:

The **Command Pattern** is a **behavioral design pattern** that encapsulates requests as objects, allowing them to be queued, logged, and executed at different times. It is widely used in **undo/redo operations, GUI button actions, and remote control implementations**.

Example: Remote Control for a Home Automation System

Imagine a smart home where **devices (Lights, Fans, AC)** can be controlled via a **remote**. Using the Command Pattern, we **encapsulate device actions (on/off) as command objects**, making the system **flexible and scalable**.

Interpreter Design Pattern:

Interpreter Design Pattern in Java

The **Interpreter Pattern** is a **behavioral design pattern** that defines a way to interpret languages, expressions, or grammars. It is useful for **parsing mathematical expressions, processing SQL queries, or implementing simple scripting languages**.

Example: Evaluating Mathematical Expressions

Imagine a system that interprets **simple mathematical expressions** like `"10 + 5"` or `"20 - 3"`. Instead of manually parsing them every time, we use **Interpreter Pattern** to define a structured approach.

Mediator Design Pattern:

Mediator Design Pattern in Java

The **Mediator Pattern** is a **behavioral design pattern** that reduces direct dependencies between objects by introducing a **mediator** that facilitates communication. It helps **decouple components** and makes the system **more maintainable**.

Example: Chat Room Mediator

Imagine a chat application where **multiple users communicate** through a **chat room**. Instead of users talking directly to each other (tight coupling), we introduce a **mediator** (`ChatRoom`) that manages communication.

Memento Design Pattern

The **Memento Pattern** is a **behavioral design pattern** that allows an object to **save and restore its previous state** without exposing its internal structure. It is commonly used for **undo/redo functionality in text editors, game save states, and transaction rollbacks**.

Example: Undo Functionality in a Text Editor

Imagine a **text editor** where users can **type text** and **undo changes**. Using the **Memento Pattern**, we allow the editor to save **snapshots** of text and restore them when needed.

Observer Method Design Pattern

The **Observer Pattern** is a **behavioral design pattern** that establishes a **one-to-many relationship** between objects, where multiple observers are notified when the subject's state changes. It is commonly used for **event handling, pub-sub messaging, and UI listeners**.

Example: Weather Monitoring System

Imagine a weather station that **notifies multiple displays** (TV, Mobile App, Website) whenever the **temperature changes**.

State Method Design Pattern

The **State Pattern** is a **behavioral design pattern** that allows an object to change its behavior **based on its internal state** without modifying its class. It helps **eliminate complex conditional logic** and makes state transitions more **maintainable**.

Example: Traffic Light System

Imagine a **traffic light system** where the signal changes between **Red, Yellow, and Green** states. Instead of handling transitions using **if-else conditions**, we use the **State Pattern** for better flexibility.

Strategy Method Design Pattern

The **Strategy Pattern** is a **behavioral design pattern** that allows selecting an **algorithm dynamically** at runtime. It helps **decouple algorithm selection** from implementation, making the system **more flexible and extensible**.

Example: Payment Processing System

Imagine an **e-commerce system** where customers can **pay using different methods** (Credit Card, PayPal, UPI). Using the **Strategy Pattern**, we allow dynamic selection of payment methods **without modifying existing code**.

Template Method Design Pattern

The **Template Method Pattern** is a **behavioral design pattern** that defines the **skeleton of an algorithm** in a base class and allows **subclasses** to override specific steps **without changing the overall structure**.

Example: Coffee Preparation Process

Imagine a **coffee machine** that brews different types of coffee (**Black Coffee, Milk Coffee**). The preparation steps (**boiling water, brewing coffee, serving**) remain the same, but specific steps (**adding milk**) vary based on the type.

Visitor Method Design Pattern

The **Visitor Pattern** is a **behavioral design pattern** that allows adding new operations to an object structure **without modifying its classes**. It helps **decouple behavior from objects**, making them more extensible.

Example: Employee Bonus Calculation

Imagine a company where employees **receive different bonuses** based on their role (**Manager, Developer**). Instead of modifying employee classes to calculate bonuses, we **use Visitor Pattern to add this behavior dynamically**.

Explain the best way to create the Singleton class?

1. Eager Initialization (Simple but may waste resources)

Java

 Copy

```
public class Singleton {
    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

- **Pros:** Thread-safe without synchronization.
- **Cons:** Instance is created even if it's never used.

2. Lazy Initialization (Not thread-safe)

Java

 Copy

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- **Pros:** Instance is created only when needed.
- **Cons: Not thread-safe**—multiple threads can create multiple instances.

3. Synchronized Method (Thread-safe but slow)

Java

 Copy

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

- **Pros:** Thread-safe.
- **Cons: Performance overhead** due to synchronization.

4. Double-Checked Locking (Best balance of safety & performance)

```
Java Copy
public class Singleton {
    private static volatile Singleton instance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
}
```

- **Pros:** Thread-safe with minimal synchronization overhead.
- **Cons:** Slightly complex.

5. Bill Pugh Singleton (Recommended)

The **Bill Pugh Singleton** is a clever way to implement a **thread-safe and efficient singleton** in Java using **static inner classes**. It avoids the need for synchronization while ensuring lazy initialization.

How It Works

The core idea is to use a **static inner helper class**, which holds the singleton instance. Java ensures that the inner class is not loaded until it's actually needed—this provides **lazy initialization** without explicit synchronization.

Implementation

```
Java Copy
public class Singleton {
    private Singleton() {}

    private static class SingletonHelper {
        private static final Singleton INSTANCE = new Singleton();
    }

    public static Singleton getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Why It's Effective

1. **Lazy Initialization:** The instance is created only when `getInstance()` is called.
2. **Thread-Safe:** Since class loading in Java is inherently thread-safe, we avoid synchronization overhead.
3. **Efficient:** No need for `synchronized` or `volatile`, making it **fast**.
4. **Encapsulation:** The helper class is only used internally, keeping the design clean.

How Java Handles It

- The **SingletonHelper class** is not loaded until `getInstance()` is called for the first time.
- The JVM ensures that the initialization of `INSTANCE` happens **only once**, making it **thread-safe**.